A Conservative Technique to Improve Deterministic Evaluation of Logic Programs*

A. Roychoudhury C.R. Ramakrishnan I.V. Ramakrishnan
Dept. of Computer Science
SUNY at Stony Brook, NY 11794.
{abhik, cram, ram}@cs.sunysb.edu

R. Sekar
Dept. of Computer Science
Iowa State University, Ames, IA 50011.
sekar@cs.iastate.edu

Abstract

The performance of logic programs can be significantly improved by reducing nondeterminism in their evaluation using techniques for early pruning of computation paths that would eventually fail. Using static information gleaned from the program, we can identify (simple) conditions that must hold for certain computation paths to succeed, and test them before searching along those paths. However, naive introduction of such tests can actually lead to performance degradation since tests may be repeated along a branch, and also because the tests themselves may create additional choice points. We therefore develop a program transformation algorithm that enables us to introduce only those tests that facilitate early pruning of failure branches, while providing formal guarantees against any performance degradation. Our transformation is based on a novel polyvariant program specialization technique that can reason about the relative execution times of the original and transformed programs. We present results of a prototype implementation that shows the effectiveness of our approach.

Keywords: *Optimization of Logic Programs, Determinacy, Program Specialization.*

1 Introduction

The ability to perform non-deterministic search is one of the most attractive and powerful features of logic programming languages such as Prolog. At the same time, in many cases, solutions are found only on a few of the possible computation paths. Significant performance gains can be obtained by identifying and avoiding searches down those paths of computation that would eventually fail to produce a solution.

One of the earliest known techniques for improving determinacy is shallow-backtracking [3, 11, 19, 27]. However, this technique cannot propagate failure-related information across predicates or program clauses. Functionality analysis [8, 9], mutual exclusion analysis [18] and cut-based determinacy analysis [16, 24] overcome this drawback. However, these techniques perform an all-or-nothing optimization: no optimization is possible if we cannot determine whether a predicate is functional, or a set of clauses are mutually exclusive. *Necessary-condition* based techniques [5, 12, 15, 23] combine the benefits of the above two categories of techniques. They exploit failure-related information that may be embedded deep within a program, and use it systematically to prune failure-bound branches, regardless of whether the branch involves deterministic or non-deterministic predicates.

Specifically, necessary-condition based techniques derive a condition for each program clause that must be satisfied in order for the clause to be used in computing a solution. At run time, a clause is selected only if the condition associated with that clause is satisfiable. Naive tests for satisfiability can however degrade performance due to repetition of tests. This problem is compounded by the fact that the satisfiability check itself may lead to creation of new choice points. Hence, a direct implementation of this model can lead to significant loss in performance [4]. A natural problem, then, is to design a program transformation technique for introducing satisfiability tests at the earliest possible point (thereby promoting early pruning) *without degrading the program's performance.* In this paper, we present a solution to this problem. Our transformation technique, called SNIP (**S**pecialization using **N**ecessary conditions to **I**mprove **P**runing), is based on polyvariant program specialization.

1.1 Overview of Approach

The input to our transformation algorithm consists of a Prolog program, together with a description of the set of all possible top-level goals, called *permissible queries.* The set of permissible queries can be compactly described by mode declarations (specifying which predicate arguments are inputs and/or outputs)

*This work was partially supported by NSF grants CCR 9404921, 9510072, 9705998, 9711386, CDA 9504275 and INT 9600598.

and export declarations (specifying which predicates are visible outside the module). We will use the following example program to illustrate SNIP.

Example 1
```
p([a]).
p([a|X]) :- p(X).
```

Consider permissible queries of the form `p(t)`, where t is any ground term. Our first step is to use a *depth-k* program analysis technique such as that described in [5] to infer the *clause condition* for each program clause. The clause condition is a constraint that must be satisfied whenever an answer to a permissible query can be computed using the clause. We annotate the program with the clause conditions; to simplify the description, we also move all the unifications in the head of a clause to its body, as shown below:

$$\texttt{p(X) :- } (\texttt{X} = [\texttt{a}],\ \phi) \mid \texttt{X} = [\texttt{a}].$$
$$\texttt{p(X) :- } (\texttt{X} = [\texttt{a}|\texttt{X1}] \wedge \texttt{X1} = [\texttt{a}|\texttt{X2}],\ \phi) \mid \texttt{X} = [\texttt{a}|\texttt{X1}],\ \texttt{p(X1)}.$$

SNIP breaks up a clause condition into a series of primitive tests, and attempts to promote these tests into the body of the program. In order to provide the necessary performance assurance, the promoted tests are such that (a) we can statically assure that the tests will not introduce additional choice points, and (b) the cost of the newly introduced test can be "absorbed" by specializing the clause body to eliminate a test with equivalent cost further down in the computation path. SNIP promotes such tests and proceeds to specialize the body of the clause (together with the predicates occurring within them) to eliminate any other tests that are implied by the newly-introduced tests. Tests that cannot be promoted are simply discarded. This process is repeated until every test is either promoted or discarded. At this point, we have transformed an input program $\mathbf{P}$ into another program $\mathbf{P}_T$ such that

- $\mathbf{P}$ and $\mathbf{P}_T$ compute the same set of answers in the same order for every permissible query, and
- $\mathbf{P}_T$ tests the necessary conditions as early as possible, while ensuring that
- $\mathbf{P}_T$ evaluates every permissible query at least as quickly as the untransformed program $\mathbf{P}$. Specifically, successful computation paths for $\mathbf{P}_T$ are no longer than the corresponding paths for $\mathbf{P}$, and, no new failure paths are introduced in $\mathbf{P}_T$.

The transformed program for our example is:

```
p([a]).
p([a,a|X]) :-  p1(X1)
p1([]).
p1([a|X]) :-  p1(X).
```

1.2 Salient Features of SNIP

1. In contrast to most specialization techniques, SNIP provides a formal assurance about the relative performance of the transformed program over that of the original program.
2. SNIP performs aggressive specialization, uniformly handling specialization contexts with disjunction. (See Section 4 for a brief discussion relating SNIP to partial evaluation.)
3. A prototype implementation of SNIP shows that the aggressive specialization strategy leads to significant performance gains, while still retaining the assurances regarding worst-case performance.

$LL(k)$ grammars provide an interesting example of the effectiveness of SNIP. Given depth-k necessary conditions [5, 23], SNIP transforms a DCG representation of an $LL(k)$ grammar into the equivalent (deterministic) $LL(k)$ parser.

1.3 Organization of the Paper

The rest of the paper is organized as follows. After stating the notational conventions and basic definitions in Section 1.4, we describe our technique, SNIP, and its effectiveness in Section 2. In Section 3, we briefly sketch the proofs of soundness and termination of our technique. Detailed comparisons of our work with earlier works on deterministic evaluation as well as partial evaluation and specialization appear in Section 4. Finally, we discuss potential extensions of our technique in Section 5.

1.4 Notations and Conventions

We use the following naming conventions. These names may sometimes be used with subscripts and superscripts:

α, β, γ	predicate clauses
Γ	the set of all terms
$\mathcal{C}$	context conditions, or simply, contexts
$\mathcal{F}$	the set of uninterpreted function symbols
f, g, h	uninterpreted function symbols
μ	program points
ω	SLD-derivations
$\mathcal{P}$	the set of all *predicate symbols*
p, q, r, s	predicate symbols
$\mathbf{P}, \mathbf{Q}$	programs
σ, θ	substitutions
t, u, v, w	terms
$\mathcal{V}$	the set of all variables
X, Y, Z	variables
$\overline{X}, \overline{Y}, \overline{Z}$	lists of variables
φ, ψ	constraints
_	anonymous variable

The notation $[X \mapsto t]$ means that X assumes the substitution t. The value of variable X under substitution σ is denoted by $\sigma(X)$. An *elementary unification operation* is of the form $X = f(\overline{X})$ where f is an *n-ary* function symbol in $\mathcal{F}$, and cardinality of $\overline{X}$ is n. Each clause in a program is of the form $p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n})$ where the unifications in the body are all elementary unification operations. Note that this form does not restrict the set of programs we consider, since all programs can be readily transformed into this form. An *elementary constraint* has the form $p(t_1, t_2, \ldots, t_n)$ where each $t_i \in \Gamma$ and $p \notin \mathcal{P}$. *Constraints* are built using conjunction and disjunction over elementary constraints. A constraint $\varphi(\overline{X})$ is parametrized w.r.t variables in $\overline{X}$. Application of substitution σ to constraint φ is denoted by $\varphi[\sigma]$.

We assume familiarity with the standard notions of SLD derivations and SLD derivation trees [14]. An *answer substitution* is the substitution computed for the variables in the goal by a successful derivation. We assume Prolog-style evaluation, and for the sake of simplicity, consider only positive programs without control features or side effects (e.g., *cut*).

2 Transformation Algorithm

We begin this section with the concepts and definitions needed to describe our algorithm. First we formalize the notion of a *context* at a program point μ which specifies the conditions that are known to hold whenever that program point is reached in any evaluation of any permissible top-level query. More formally,

Definition 1 (Context) *Let ω be a derivation starting with a permissible top-level query, and let σ_ω^μ be the substitution computed by this derivation at a program point μ. Then the context at μ is a constraint $\mathcal{C}_\mu$ such that $\mathcal{C}_\mu[\sigma_\omega^\mu]$ is true, for all such ω.*

The following notion of *clause condition* is stronger than the notion of *context* in that it also takes into account those conditions that would be tested after the computation reaches a certain program point (the point where the clause is invoked).

Definition 2 (Clause-Condition) *A constraint φ is a clause-condition for a clause α iff the following holds for any successful derivation ω starting with any permissible top-level query: if α is used in ω then $\varphi[\sigma_\omega]$ is true (where σ_ω is the substitution computed by ω).*

Note that clause conditions are *constraints*, whereas we can introduce (into programs) only primitive operations or *tests* that can be evaluated by a Prolog engine.

Definition 3 (Test and Elementary Test) *An elementary test is either a program builtin or an elementary unification operation. A* test *is a sequence of elementary tests.*

We also need a formal way to map from tests to constraints.

Definition 4 (Success-constraint) *The success-constraint of a built-in predicate built_in($\overline{X}$) is a constraint ψ(built_in($\overline{X}$)) that holds whenever built_in($\overline{X}$) succeeds.*

In our transformation algorithm, we annotate each clause (in the program being transformed) with the corresponding clause conditions and also those tests that have been promoted into the clause body from the clause condition:

Definition 5 (Annotated Program) *An annotated program consists of clauses of the form* `q(X̄) :-` $\mathcal{N}|\mathcal{B}$ *where $q(\overline{X})$ is the head, $\mathcal{N}$ is the neck and $\mathcal{B}$ is the body. $\mathcal{N}$ consists of the clause-condition φ and the promoted tests* $\mathbf{D}$.

2.1 Algorithm *Transform*

The top-level procedure in the transformation algorithm is called *Transform* which takes an annotated program $\mathbf{P}$ as its argument and returns the transformed program $\mathbf{P}_t$. It iterates through the clauses in $\mathbf{P}$ and moves as many tests from clause conditions into clause bodies as possible. The actual movement is performed by a second level procedure called *IntroduceTest*.

Transform($\mathbf{P}$) returns $\mathbf{P}_t$
1. $\mathbf{P}_t := \mathbf{P}$
2. while $\mathbf{P}_t$ contains a clause α that is not marked "done"
3. $\mathbf{P}_t := IntroduceTest(\alpha, \mathbf{P}_t)$
4. return $\mathbf{P}_t$

To provide guarantees on the performance of the transformed program, *IntroduceTest* ensures that the cost of the newly introduced test is compensated by elimination of the same (or equivalent) test(s) further down in the program. This elimination is achieved by a transformation process that specializes the body of the current clause and the (definitions of the) predicates used therein. The transformed program is returned by *IntroduceTest*. If no new tests could be introduced in α, then *IntroduceTest* marks the clause as "done." Marked clauses are not considered again for test introduction.

We illustrate *Transform* using the following example.

α_1 : `p(X) :- (X = [a], ϕ) | X = [a].`
α_2 : `p(X) :- (X = [a|X1] ∧ X1 = [a|X2], ϕ) |`
`X = [a|X1], p(X1).`

Transform will first invoke *IntroduceTest* on α_1. *IntroduceTest* attempts to introduce the test $\mathtt{X} = [\mathtt{a}]$ from the neck to the body, and the cost of introduction is paid for by elimination of the same test. Thus we get the (trivial) transformation to a new clause:
α_3: `p(X) :-` $(\mathtt{X} = [\mathtt{a}], \{\mathtt{X} = [\mathtt{a}]\}) \mid \mathtt{X} = [\mathtt{a}]$.
Now the program consists of the clauses $\{\alpha_3, \alpha_2\}$. In the next iteration through the loop in *Transform*, *IntroduceTest* is invoked on α_3. Since no more tests can be introduced into the body of this clause, *IntroduceTest* returns without any further transformation, but simply marking α_3 as done. *Transform* then invokes *IntroduceTest* on α_2, which performs the (trivial) step of introducing the test `X=[a|X1]` into the clause body and eliminating it. We now have
α_4: `p(X) :-` $(\mathtt{X} = [\mathtt{a}|\mathtt{X1}] \wedge \mathtt{X1} = [\mathtt{a}|\mathtt{X2}], \{\mathtt{X} = [\mathtt{a}|\mathtt{X1}]\}) \mid$
$\mathtt{X} = [\mathtt{a}|\mathtt{X1}], \mathtt{p}(\mathtt{X1})$.
Now *IntroduceTest* is invoked on α_4. *IntroduceTest* moves `X1 = [a|X2]` into the clause body. It then tries to eliminate tests implied by the newly introduced tests. As we will explain later, this elimination is achieved by replacing the call `p(X1)` in the clause body by a call `p1(X2)` to a specialized version of `p`. By passing only the unexamined portion `X2` of `X1` into `p1`, we avoid reexamining the structure of `X1`. At this point, α_4 is replaced by the following clauses:

α_5: `p(X) :-` $(\mathtt{X} = [\mathtt{a}|\mathtt{X1}] \wedge \mathtt{X1} = [\mathtt{a}|\mathtt{X2}],$
$\{\mathtt{X} = [\mathtt{a}|\mathtt{X1}], \mathtt{X1} = [\mathtt{a}|\mathtt{X2}]\}) \mid$
$\mathtt{X} = [\mathtt{a}|\mathtt{X1}], \mathtt{X1} = [\mathtt{a}|\mathtt{X2}], \mathtt{p1}(\mathtt{X2})$.
$\alpha_6 : \mathtt{p1}(\mathtt{X2}) :- \ (\mathtt{X2} = [\ ], \phi) \mid \mathtt{X2} = [\ ]$.
$\alpha_7 : \mathtt{p1}(\mathtt{X2}) :- \ (\mathtt{X2} = [\mathtt{a}|\mathtt{X3}], \phi) \mid \mathtt{p}(\mathtt{X2})$.

Finally, we attempt to introduce the test `X2 = [a|X3]` into the body of α_7 and then create a specialized version of `p(X2)` for the context `X2 = [a|X3]`. Noting that the previous specialization of `p` was for the same context, we finally obtain the following clause in place of α_7:
α_8: `p1(X2) :- (X2 = [a|X3], {X2 = [a|X3]}) |`
`X2=[a|X3], p1(X3).`
Also, we introduce the test `X2 = [ ]` in α_6 to get
$\alpha_9 : \mathtt{p1}(\mathtt{X2}) :- \ (\mathtt{X2} = [\ ], \{\mathtt{X2} = [\ ]\}) \mid \mathtt{X2} = [\ ]$.
The final program consists of clauses $\alpha_3, \alpha_5, \alpha_8$ and α_9. Observe that the transformed program can be executed in a deterministic fashion by any Prolog engine that uses deep indexing (such as XSB), whereas evaluation of the original program requires backtracking.

2.2 Algorithm *IntroduceTest*

The actual task of introducing a test from a clause condition into the clause body is performed by *IntroduceTest*. This algorithm uses a function *Select* to identify tests that can be promoted into clause bodies from clause conditions. For each test thus identified, it uses another procedure *AbsorbTest* to perform specialization aimed at eliminating equivalent tests that may be performed in the clause body or within the predicates invoked from the clause. At first sight, it may appear that almost any test selected from the clause condition can be introduced (and the clause body specialized) in this manner, without causing overall execution times to increase. This is because the clause condition consists of *necessary conditions* that must be checked directly or indirectly in every successful execution path using this clause. However, several complications arise

- As mentioned earlier, necessary conditions are *constraints,* whereas we can introduce only *tests* into the program that can be evaluated at the point they appear.

- The clause condition may contain disjunctions such as $(X = f(b,c)) \vee (X = f(c,Z))$. A natural way to deal with this situation is to identify a set of tests $\{t_1, \ldots, t_n\}$ such that (the clause condition implies that) one of these tests are performed in every successful derivation. Then we can create n specialized versions of the current clause, where t_i is promoted into the ith clause. However this transformation may create an n-way choice point where none existed before. To avoid this possibility, we must first ensure that the t_i's are mutually incompatible. But this alone is not enough, since the tests $(X = f(b,c))$ and $(X = f(c,Z))$ are mutually incompatible, but both are compatible with a goal substitution $X = f(U,c)$. On the other hand, if we know that $X = f(U,V)$ where U is bound, we can again ensure that at most one of the two tests can be satisfied.

- A newly introduced test can alter the bindings associated with variables in such a way that costs of subsequent operations are increased (*e.g.* by converting a binding to a matching).

- Even if the promoted test satisfies the above three conditions, its promotion may still increase overall costs because we are unable to eliminate tests with equivalent costs below — in spite of the fact that the promoted test is a necessary condition. In particular, since the computation of clause conditions involves approximations, it is possible that the promoted test t itself is not tested below this point, but only a test t' that is strictly stronger than t.

Thus, in order to provide formal guarantees, we need to ensure that the tests returned by *Select* satisfy the above conditions. We then attempt to absorb the cost of the promoted test through specialization. If this attempt is unsuccessful, we proceed to select alternative tests for introduction. The formal definition of *IntroduceTest* is given as :

$IntroduceTest(\alpha, \mathbf{P})$ returns $\mathbf{P}_t$
1. Let α be of the form $q^{\mathcal{C}}(\overline{X}) :- (\varphi, \mathbf{D}) | \mathcal{B}$;
2. while $((\{t_1, ..., t_n\} \equiv Select(\varphi, \mathcal{C}, \mathbf{D})) \neq nil)$
3. $\quad \mathbf{P}' := \mathbf{P}$
4. $\quad$ for $i := 1$ to n do
5. $\quad\quad (\mathcal{B}', \_, \mathbf{T}', \mathbf{P}') :=$ $AbsorbTest(\mathcal{B}, \mathcal{C} \wedge t_i, \{t_i\}, \mathbf{P}')$
6. $\quad\quad$ if $\mathbf{T}'$ is empty then $\mathbf{P}' := \mathbf{P}' \cup \{q^{\mathcal{C}}(\overline{X}) :- (\varphi, \mathbf{D} \cup \{t_i\}) \mid t_i, \mathcal{B}'\}$
7. $\quad\quad$ else
8. $\quad\quad\quad \mathbf{D} := \mathbf{D} \cup \{t_i\}$
9. $\quad\quad\quad$ continue while
10. $\quad$ endfor
11. $\quad$ return $\mathbf{P}' - \{\alpha\}$
12. endwhile
13. Mark α as done.
14. return $\mathbf{P}$

Given a Prolog program $\mathbf{P}$ and a permissible query G we obtain the corresponding annotated program by replacing every clause α of the form $q(\overline{X})$:- $\mathcal{B}$ by the clause $q^{true}(\overline{X})$:- $(\varphi, \phi) \mid \mathcal{B}'$ where φ is the clause condition of α, and $\mathcal{B}'$ is obtained from $\mathcal{B}$ by annotating the occurrence of all user-defined predicates with the context $true$. We also add a clause $top^{\mathcal{C}}(\overline{X})$:- $p^{true}(\overline{X})$ where *top* does not appear in $\mathbf{P}$ and G is the instantiation of $p(\overline{X})$ under context $\mathcal{C}$[1]. To avoid clutter, in the illustrations of the paper, we explicitly specify the context $\mathcal{C}$ of a specialized predicate $p^{\mathcal{C}}$ in words. Also, we avoid showing the clause of the form $top^{\mathcal{C}}(\overline{X})$:- $p^{true}(\overline{X})$ that we add while constructing any annotated program.

Now, in algorithm *IntroduceTest*, the function *Select* takes the current context $\mathcal{C}$ as an argument so that it can return only tests that can be evaluated in this context. It also takes the set $\mathbf{D}$ of tests that we have already attempted to introduce into this clause so as to avoid returning the same test over and over. We do not specify an implementation of *Select* here ,- *IntroduceTest* is parameterized w.r.t. to this function, enabling us to change the behavior of *IntroduceTest* by changing *Select*. The conditions that need to be satisfied by *Select* in order to provide formal guarantees are discussed in section 2.4. In general, *Select* may not return a single test from the clause condition, but a set of tests $\{t_1, \ldots, t_n\}$. We then need to generate n specialized clauses from the current clause such that t_i is introduced into the ith specialization (polyvariant specialization: done in loop at lines 4–10). If t_i can be absorbed, then the new (specialized) clause is added to the current program $\mathbf{P}'$ at line 6. Otherwise, if any of the n tests cannot be absorbed then we try to go back and select alternative tests. To ensure that a test that has been selected once for introduction is not selected again, we update $\mathbf{D}$ at lines 6 and 8.

To illustrate *IntroduceTest*, consider the program

Example 2
```
ans(Y,Z) :- costly(Y,Z), q(Y).
q(b).
q(c).
```

Let the annotated version of the first clause be

$$\texttt{ans(Y,Z)} :- \ (\texttt{Y} = \texttt{b} \vee \texttt{Y} = \texttt{c}, \phi) \ \mid \ \texttt{costly(Y,Z)}, \texttt{q(Y)}.$$

and also assume that `ans` is called with its first argument ground, denoted $Y \in g$. Then $Select(Y = b \vee Y = c, Y \in g, \phi)$ returns $\{Y = b, Y = c\}$. Observe that, based on the context, both the tests can be evaluated at the time `ans` is invoked. The clause generated by introduction of `Y=b` is:

```
ans(Y, Z) :-  (Y=b ∨ Y=c, {Y=b})  |
              Y = b, costly(Y,Z), q1.
```

where `q1` is obtained by specializing `q` given that its argument is `b`. Note that in this specialized version, we can avoid the test `Y=b` that would otherwise have been performed in the body of `q`. Thus the cost of promoting the test has been absorbed. Similarly, the introduction of `Y=c` yields the specialized clause:

```
ans(Y, Z) :- (Y=b ∨ Y=c, {Y=c})  |
             Y = c, costly(Y,Z), q2.
```

where we have again been able to eliminate the test `Y=c` that would have been performed in evaluating `q`. Thus we can absorb the newly introduced test along both branches of computation, so we would retain these transformed clauses and discard the original clause.

2.3 Algorithm *AbsorbTest*

The purpose of *AbsorbTest* is to specialize the body of a clause and the literals contained in the clause. The specialization is based on the context information $\mathcal{C}$ that captures information that is known about the variables appearing in the clause. While performing specialization, we check if some of the tests $\mathbf{T}$ that have been newly introduced before this clause can be eliminated in the specialized versions. It returns the transformed program and the subset of tests that have not been absorbed in this manner.

AbsorbTest is defined using the equations below. Its structure is simple: it loops through the literals in the

[1]The context of all other predicates will be propagated in the course of our transformation.

body of the clause, delegating the task of specializing each of these literals to the function *AbsorbLit*. The set of tests yet to be absorbed, the current context and program are all "updated" as we specialize the literals, by threading these arguments through successive invocations.

$$
\begin{aligned}
&\mathit{AbsorbTest}(\mathit{nil},\mathcal{C},\mathbf{T},\mathbf{P}) = (\mathit{nil},\mathcal{C},\mathbf{T},\mathbf{P})\\
&\mathit{AbsorbTest}((r(\overline{X}),\mathcal{B}),\mathcal{C},\mathbf{T},\mathbf{P}) =\\
&\quad \text{Let}\quad (r'(\overline{X}'),\mathcal{C}',\mathbf{T}',\mathbf{P}') =\\
&\qquad\qquad \mathit{AbsorbLit}(r(\overline{X}),\mathit{project}(\mathcal{C},\overline{X}),\mathbf{T},\mathbf{P})\\
&\qquad\quad (\mathcal{B}',\mathcal{C}'',\mathbf{T}'',\mathbf{P}'') =\\
&\qquad\qquad \mathit{AbsorbTest}(\mathcal{B},\mathcal{C}'\wedge\mathcal{C},\mathbf{T}',\mathbf{P}')\\
&\quad \text{in}\quad ((r'(\overline{X}'),\mathcal{B}'),\mathcal{C}'',\mathbf{T}'',\mathbf{P}'')
\end{aligned}
$$

The function *AbsorbLit* takes four arguments : (a) the literal $r(\overline{X})$ to be specialized, (b) the context in which this literal will be evaluated. This context is obtained by projecting the current context in *AbsorbTest* onto the arguments $\overline{X}$ of r, (c) the tests $\mathbf{T}$ yet to be absorbed through specialization, and (d) the current program $\mathbf{P}$ It returns the specialized literal $r'(\overline{X}')$, which refers to a specialized version of r. It also returns a new context that captures the conditions that hold after the evaluation of this literal, the subset $\mathbf{T}'$ of $\mathbf{T}$ that were not absorbed through the specialization, and the transformed program $\mathbf{P}'$.

The first equation defining *AbsorbLit* (see below) deals with the case when the literal to be specialized is a built-in. The function *Remove* is used to simplify (or even eliminate) this built-in based on the current context. If this simplification results in cost reductions that can offset the cost of some of the promoted tests $\mathbf{T}$, such tests are removed from $\mathbf{T}$ by *Remove* to get the set $\mathbf{T}'$ of tests that are yet to be absorbed. Finally, the context $\mathcal{C}$ is updated to indicate the fact that the test $\mathit{built\_in}(\overline{X})$ has been performed.

$$
\begin{aligned}
&\mathit{AbsorbLit}(\mathit{built\_in}(\overline{X}),\mathcal{C},\mathbf{T},\mathbf{P}) =\\
&\qquad (\mathit{built\_in}'(\overline{X}'),\mathcal{C}\wedge\psi(\mathit{built\_in}(\overline{X})),\mathbf{T}',\mathbf{P})\\
&\text{where } (\mathit{built\_in}'(\overline{X}'),\mathbf{T}') = \mathit{Remove}(\mathbf{T},\mathit{built\_in}(\overline{X}),\mathcal{C})
\end{aligned}
$$

The second equation for *AbsorbLit* (see below) is applicable when r is user-defined. In this case, we first identify the subset of clauses defining r that are applicable in the current context, and specialize their bodies using the function *AbsorbClauses*. We then introduce a new version $r^{\mathcal{C}}$ of r whose definition is given by these bodies. Its arguments are computed by using a function called *depend*. Intuitively, the arguments of $r^{\mathcal{C}}$ consist of all variables Y for which the following conditions hold: *(a)* instantiation of Y depends on some variable in $\overline{X}$ whenever $\mathcal{C}$ is true *(b)* Y is used deeper down in at least one of the computation paths of $r(\overline{X})$. Hence in the specialized version $r^{\mathcal{C}}$, we try to pass as arguments only unexamined subterms of the structures which appear as arguments of r[2].

$$
\begin{aligned}
&\mathit{AbsorbLit}(r(\overline{X}),\mathcal{C},\mathbf{T},\mathbf{P}) =\\
&\quad \text{Let } \mathbf{R} \text{ be the clauses defining } r \text{ whose}\\
&\qquad\quad \text{clause conditions are compatible with } \mathcal{C}\\
&\qquad (\mathbf{R}',\mathcal{C}',\mathbf{T}',\mathbf{P}') = \mathit{AbsorbClauses}(\mathbf{R},\mathcal{C},\mathbf{T},\mathbf{P})\\
&\qquad \overline{X}' = \textstyle\bigcup_{R'\in\mathbf{R}'} \mathit{depend}(\overline{X},\mathcal{C},R'),\\
&\quad \text{in } (r^{\mathcal{C}}(\overline{X}'),\mathcal{C}',\mathbf{T}',\mathbf{P}'\cup\{r^{\mathcal{C}}(\overline{X}') : -R' | R'\in\mathbf{R}'\})
\end{aligned}
$$

To complete the algorithm, we now provide the rules for *AbsorbClauses*. It iterates through all of the clauses in its first argument, and specializes the body of each of them using *AbsorbTest*.

$$
\begin{aligned}
&\mathit{AbsorbClauses}([\,],\mathcal{C},\mathbf{T},\mathbf{P}) = ([\,],\mathit{false},\mathit{false},\mathbf{P}).\\
&\mathit{AbsorbClauses}([R|\mathbf{R}],\mathcal{C},\mathbf{T},\mathbf{P}) =\\
&\quad \text{Let } \mathcal{N} \text{ be the neck of } R \text{ and } \mathcal{B} \text{ its body}\\
&\qquad (\mathcal{B}',\mathcal{C}',\mathbf{T}',\mathbf{P}') = \mathit{AbsorbTest}(\mathcal{B},\mathcal{C},\mathbf{T},\mathbf{P})\\
&\qquad (\mathbf{R}',\mathcal{C}'',\mathbf{T}'',\mathbf{P}'') = \mathit{AbsorbClauses}(\mathbf{R},\mathcal{C},\mathbf{T},\mathbf{P}')\\
&\qquad R' = \mathcal{N}|\mathcal{B}'\\
&\quad \text{in } ([R'|\mathbf{R}'],\mathcal{C}'\vee\mathcal{C}'',\mathbf{T}'\cup\mathbf{T}'',\mathbf{P}'')
\end{aligned}
$$

Note that *AbsorbTest* returns the new context that holds at the end of evaluating the specialized clause, and also the tests that have not yet been absorbed. The context that holds after evaluation of any one of these specialized clauses is simply the disjunction of the contexts returned by *AbsorbTest* for each of the clauses. Similarly, the tests yet to be absorbed at the end of *AbsorbClauses* includes all the tests that may not be absorbed in one of the clauses.

2.4 Requirements on *Select* and *Remove*

Select: Whenever $\mathit{Select}(\varphi,\mathcal{C},\mathbf{D})$ returns a set of tests $\{t_1,...,t_n\}$ the following conditions must hold:
Soundness of Specialization: $\varphi \Rightarrow \bigvee_{i=1}^{n} t_i$
Avoiding choice points : $\forall\sigma$ that satisfy $\mathcal{C}$ $\forall\beta,\gamma$ that are instances of σ $\forall i\neq j$ $\neg(t_i\beta\wedge t_j\gamma)$
Nonredundancy: $\forall i$ $[(\varphi\wedge\mathcal{C}\wedge t_i)$ is satisfiable$]\wedge[t_i\notin\mathbf{D}]\wedge[\mathcal{C}\not\Rightarrow t_i]$
The first condition ensures that the new tests introduced do not prune away success paths and is thus required for soundness. Given any substitution σ that satisfies the context $\mathcal{C}$, the second condition ensures that it is *impossible* to instantiate σ one way to take the ith specialized clause, backtrack, and then come back to take another jth specialized clause. The third condition ensures that we do not select redundant tests that are (a) incompatible with the clause condition and

[2]A subterm which has already been examined is passed only if it is used deeper down. This is to avoid the overhead of redundant term construction.

Benchmarks	CPU time (Original pgm.)	CPU time (after naive insertion)	CPU time (Final pgm.)	Speedup	Code-size increase
LL(2)	20.18	$\geq$ 30,000	14.32	1.41	1.68
ListtoAtom	10.15	9.37	8.24	1.23	1.67
TreeParser	23.86	8.69	2.44	9.78	2.13
Quicksort	24.1	19.13	18.55	1.3	2.18
DNA Parser	67.62	--	19.37	3.49	7.5

Table 1: Performance improvement through determinacy extraction

the context, or (b) have already been selected, or (c) implied by the context.

In the actual implementation, we can satisfy *soundness of specialization* by choosing a conjunct ψ_i from each conjunction in φ (the clause condition φ is maintained in disjunctive normal form). *Avoidance of choice points* is satisfied by choosing tests on variables whose inferred modes are ground and by also ensuring that $\forall i, j,\ \psi_i \wedge \psi_j$ is not satisfiable if $i \neq j$.

Remove: Whenever $Remove(\mathbf{T}, b(\overline{X}), \mathcal{C})$ returns $(b'(\overline{X}'), \mathbf{T}')$ (where b and b' are program builtins) the following conditions hold:

Soundness of Removal: $\mathbf{T}' \subseteq \mathbf{T}$

Soundness of Specialization: $\forall\sigma$ that satisfy $\mathcal{C}$ $(b'(\overline{X}')\sigma \Leftrightarrow b(\overline{X})\sigma)$

Soundness of Test Absorption: $\forall\sigma \mathcal{C}[\sigma] \Rightarrow \forall\beta((\beta \supseteq \sigma) \wedge (\beta \Rightarrow t_i)) \Rightarrow Cost(\mathbf{T}, \beta) - Cost(\mathbf{T}', \beta) \leq Cost(b(\overline{X}), \sigma)$

The first requirement is needed for soundness. The second requirement ensures that provided the context $\mathcal{C}$ is true, specialization of the builtin produces another builtin which succeeds or fails under the same circumstances. In the third requirement, t_i denotes the most recently introduced test. This requirement ensures that our estimate of costsavings (*i.e.*, difference in costs of $\mathbf{T}$ and $\mathbf{T}'$) correctly captures the gains achieved by replacing b with b' for any possible substitution σ compatible with $\mathcal{C}$.

2.5 Experimental Results

We implemented a prototype of SNIP using the XSB tabled logic programming system [26]. The annotated program (input to SNIP) is automatically generated by an implementation of the analysis technique of [5]. We encoded the rules of *AbsorbTest*, *AbsorbLit* and *AbsorbClauses* as a logic program, and used the tabled resolution strategy of XSB to directly compute the fixed points of these equations.

Table 1 summarizes the timings obtained before and after our program transformation was applied, as well as the increase in code size. We also indicate the timings obtained by naively introducing tests from the clause conditions into the program clauses. All measurements were taken using XSB version 1.6.1 on a SPARC Station 20 with 64 MB main memory running SunOS 5.3. All the timings given below are in CPU seconds.

$LL(2)$ is a parsing program obtained from the DCG specification of the language $(aa)^n(ab)^n$. *Treeparser* is a recursive descent parser over tree grammars. *ListtoAtom* is a predicate taken from the XSB compiler that reflects the structure of Example 1. The *DNA Parser* [1], is a Prolog program for identifying the '*introns*' or coding regions from a given DNA sequence[3]. Observe from the table that SNIP improves the performance of this program by more than a factor of three. This indicates that even while giving assurances about the worst case behavior, SNIP can effectively achieve early pruning for large programs.

3 Correctness and Performance Guarantees

The proof of correctness and performance guarantee are based on a mapping from the SLD-derivations of the original program to those of the transformed program. This mapping ensures that (a) the transformed program computes the same answers as the original program, (b) the answers are computed in the same order, and (c) every successful derivation in the transformed program is no longer than the corresponding derivation in the original program. In order to formally state these criteria, we develop the notion of *Resolution trees* which are SLD trees with cost annotations.

Definition 6 (Resolution Tree) *The resolution tree of a goal G w.r.t. a program P is the SLD-tree for G with the following annotations:* (i) *Each leaf of the tree is either an empty node (denoting success) or a*

[3]The program was given to us by Jacques Cohen of Brandeis University.

fail node. and (ii) *Any edge $N \to N'$ from node N to node N' is labelled by the operations (e.g., builtins and unification operations) performed in going from N to N'. For a root-to-leaf path E, $leaf(E)$ denotes its leaf and $cost(E)$ denotes the sum of the costs of all the operations on it.*
The notation $Succ(\mathbf{P}, G)$ denotes the set of all successful root-to-leaf paths in the resolution tree.

Note that Prolog evaluation of the goal G would correspond to a preorder traversal of the resolution tree.

Our concept of a sound program transformation is based on the notion of *similarity mapping*:

Definition 7 (Similarity Mapping) *A similarity mapping $\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G')}$ is a mapping from $Succ(\mathbf{P}, G)$ to $Succ(\mathbf{P}', G')$ (where G' is a specialization of G) s.t. :* (a) *$\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G')}$ is one-to-one and onto,* (b) *$\forall E \in Succ(\mathbf{P}, G)$, the answer substitutions of E and $\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G')}(E)$ are identical, and* (c) *$\forall E_1, E_2 \in Succ(\mathbf{P}, G)$, if i leaves are encountered between $leaf(E_1)$ and $leaf(E_2)$ during a preorder traversal of the resolution tree of G in $\mathbf{P}$, then at most i leaves are encountered between $leaf(\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G')}(E_1))$ and $leaf(\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G')}(E_2))$ in a preorder traversal of the resolution tree of G' in $\mathbf{P}'$.*

Soundness of transformation can be formally stated in terms of similarity mapping :

Definition 8 (Sound Transformation) *A program $\mathbf{P}'$ is a sound transformation of $\mathbf{P}$ for a goal G iff there exists the mapping $\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G)}$.*

That the transformed program $\mathbf{P}'$ is no worse than $\mathbf{P}$ can be formalized using the notion of:

Definition 9 (Bounded Success Paths) *Let $\mathbf{P}'$ be a sound transformation of $\mathbf{P}$ for a goal G. This transformation is said to bound the success paths in $\mathbf{P}$ denoted $\mathbf{P}' \preceq_G \mathbf{P}$, iff the following condition holds : $\forall E \in Succ(\mathbf{P}, G)$, $cost(\mathcal{I}_{(\mathbf{P},G)}^{(\mathbf{P}',G)}(E)) \leq cost(E)$.*

We now proceed to establish that our transformation meets the above mentioned conditions for soundness and performance improvement. conditions. The main component of our proof is in establishing these results for $AbsorbTest$, $AbsorbLit$ and $AbsorbClauses$. We make use of the usual fixed point construction approach for these proofs.

Termination: The termination of the transformation algorithm is guaranteed by the following theorem (proof appears in [20]).

Theorem 1 *Let $\mathbf{P}$ be any annotated logic program. Then the computation $Transform(\mathbf{P})$ terminates in a finite number of steps.*

Performance Guarantee: We establish that a test that is successfully introduced into a body, will never be repeated deeper down.

Theorem 2 (Performance Guarantee) *Let $\mathbf{P}$ be an annotated program with a permissible query G and $\mathbf{P}' = Transform(\mathbf{P})$. Then $\mathbf{P}' \preceq_G \mathbf{P}$*

Proof sketch: Since our algorithm terminates, therefore $\mathbf{P}'$ is obtained from $\mathbf{P}$ by finite number of invocations of $IntroduceTest$, *i.e.*, there is a finite sequence $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ where : $\mathbf{P} \equiv \mathbf{P}_0$, $\mathbf{P}' \equiv \mathbf{P}_n$, and $\mathbf{P}_{i+1} = IntroduceTest(\alpha, \mathbf{P}_i)$, where α is a clause in $\mathbf{P}_i$. Then, it is sufficient to prove that $\forall q^{\mathcal{C}} \in PredSet(\mathbf{P}_i)$ and $\forall\sigma$ that satisfy $\mathcal{C}$, $\mathbf{P}_{i+1} \preceq_{q^{\mathcal{C}}(\overline{X})\sigma} \mathbf{P}_i$, where $PredSet(\mathbf{P}_i)$ denotes the set of user-defined predicates in $\mathbf{P}_i$. Assuming $\alpha \equiv q^{\mathcal{C}}(\overline{X}) :- (\varphi, \mathbf{D}) \mid \mathcal{B}$, $q^{\mathcal{C}}$ is the only predicate in $PredSet(\mathbf{P}_i)$, whose success paths are altered. We then prove by induction (on the number of fixed point iterations) that whenever $AbsorbTest(\mathcal{B}, \mathcal{C}, \mathbf{T}, \mathbf{P})$ returns (_,_,nil,_), the cost of all the tests in $\mathbf{T}$ have been absorbed. Hence the cost of root-to-leaf success paths of $q^{\mathcal{C}}$ also do not increase in $\mathbf{P}_{i+1}$. The complete proof appears in [20].

4 Related Work

Shallow determinacy: The works of Carlsson [3], Hickey and Mudambi [11], Van Roy et al. [19], and Zhou et al. [27] eliminate shallow backtracking— backtracking due to failure caused by a built-in literal in the beginning of a clause body. Typically, these techniques perform built-in operations (present as the first literals in the body) as a part of indexing for selecting clauses at each resolution step. These techniques do not propagate, nor can exploit, information not immediately available in the clause being optimized.

Cardinality-based techniques: Mellish [16] and Sawamura and Takeshima [24] describe analysis methods, based on *cuts* in the program, to determine if a query to a predicate has more than one solution. Debray and Warren [8] describe a technique to detect *functional predicates* in a program— predicates that have at most one answer to every query. *Cardinality analysis* by Braem et al. [2] extends functionality analysis by estimating the number of answers to queries (instead of whether is more or less than 1). The determinacy analysis implemented in Mercury compiler [10] also estimates the cardinality of each query. Non-failure analysis proposed by Debray and

Hermenegildo [7] determines the set of goals that cannot fail: *i.e.*, have at least one answer. Mutual exclusion analysis described by Post [18] infers predicates with the property that at most one clause becomes applicable at clause selection time.

All the above methods attempt to classify the predicates in a program into multiple categories, such as functional and non-functional. The cardinality information can be used for automatic cut-insertion to reduce backtracking. However, when a predicate cannot be inferred as determinate, no further optimization is possible.

Necessary condition-based techniques: Techniques proposed by Sato and Tamaki [23], and Dawson et al. [5] infer, at compile-time, the necessary conditions for clauses to succeed. However, they do not address the central problem underlying SNIP: of effectively optimizing programs based on this information. A similar approach is used to optimize evaluation of constraint logic programs; detailed comparison with these works is given later in this section.

Note that the strategy to eagerly prune failure paths, by its very nature, enables optimization of programs even when none of the predicates are strictly *determinate*. Moreover, these techniques propagate information from deeper levels of the search tree in order to prune failure paths early, and hence can be naturally generalized to include various cardinality-based analyses described above.

"Early Failure" in CLP: Necessary condition-based techniques have been used in the context of CLP, to optimize operations over the constraint store [12, 15]. Kemp and Stuckey [12] describe a technique to push constraint selections to achieve early failure by extending the earlier work of Ramakrishnan and Srivastava [25]. They also use techniques proposed by Mariott and Stuckey [15] to remove redundant operations due to the newly introduced constraints.

The techniques employed in [12, 15] start with a source program, generate an intermediate program in which constraints are eagerly introduced and finally eliminate constraints that can be shown to be redundant. Note that introduction of constraints may add operations that were not present in the original program in the first place, and hence redundancy removal does not provide any assurance about the relative performance of the resultant program. In contrast, SNIP takes a conservative approach where tests are introduced only if they can be paid back by the elimination of equivalent tests deeper down in the search tree. It must, however, be noted that, in the evaluation of CLP programs, the gains achieved through early testing can be substantial. On the other hand our conservative approach is more appropriate for top-down evaluation of Prolog programs, since redundant testing is indeed a factor that can lead to performance degradation.

Moreover, unlike in these works, the *necessary conditions* considered by SNIP may contain disjunctions, which enables more aggressive optimization in some cases. For instance, consider the program in Example 2. The condition for success of `q(Y)` is given by $Y = b \vee Y = c$. The polyvariant specialization of SNIP generates two different versions of `q/1`. Consequently, we can promote the tests on `Y`, and avoid the unnecessary computation associated with `costly/2`. In contrast, the techniques described in [12, 15] consider only the glb of call and answer constraints, and hence do not promote the tests on `Y`.

Early Pruning by Partial Evaluation: Some of the effects of eagerly pruning failure-bound computations can also be achieved by partial evaluation. Consider a generic partial evaluator such as *Mixtus* [21, 22]. Mixtus's "left propagation of bindings" converts say `p(X,X), X = term` to `p(term,term)`. Note that this operation corresponds to our notion of test promotion. However, partial evaluators do not consider the relative costs of the resultant program. For example, consider the following predicate `get/4`, and its partial evaluation with `(ground,ground,free,free)` as the calling mode:

```
get(Z1,Z2,X,Y) :- X = Y, X = f(Z1), Y = f(Z2).
```
$\Longrightarrow$
```
get(A,A,f(A),f(A)).
```

While the original definition of `get/4` shares the answer substitution for `X` and `Y`, the modified definition results in building two different (but equivalent) terms.

Moreover, most of the current partial evaluation techniques, including *conjunctive partial evaluation* [13] specialize each computation path in isolation. This restricts the class of programs that can be optimized[4], as explained below. Consider, for instance, the program in Example 1 (page 2). Note that, for any call to `p/1` with a ground argument, at most one of the two clauses succeed. However, reasoning about this mutual exclusion needs ability to combine the conditions for success of different computation paths. In contrast to SNIP, partial evaluation techniques cannot extract and exploit this information.

5 Discussion

Promoting early failure of unsuccessful computations is a powerful optimization for enhancing deterministic evaluation of logic programs. In this paper, we

[4]Recent extensions by Pettorossi et al. [17] relax this restriction somewhat by allowing disjunctive definitions of newly defined predicates.

have presented SNIP, a technique to effectively prune failure paths by using necessary conditions. Below we discuss some possible extensions and refinements.

First of all, while SNIP takes into account the costs of any test introduced, it does not consider the benefits accrued by performing those tests, *e.g.*, possible elimination of choice-points. The underlying cost model can be easily extended to account for such benefits as well.

Secondly, we considered only those tests that do not bind program variables. Promoting operations that create variable bindings can in general increase the cost of other operations. Note that, to reason about relative costs of original and transformed programs when costs of individual operations can increase, we need upper bound estimates of the number of times loops in the program will be traversed. Such estimates can be obtained by adapting techniques from cost analysis [6]. Integrating the results of such analyses with SNIP is a topic of further research.

Finally, since SNIP performs polyvariant specialization, the code-size of the transformed program can blow up. (Observe the code size of the optimized DNA program in Section 2.5.) However, a specialized version of a predicate is generated for each context. In many cases, we can bound the number of possible contexts. For instance, for $LL(k)$ grammars, the number of contexts are limited by the size of the parsing table. Thus using SNIP we can obtain a deterministic parser for an $LL(k)$ grammar with code space linear in the size of the $LL(k)$ parsing table. Furthermore, in the fixed point computation of $AbsorbTest$, we obtain a sound program at the end of every iteration, and hence it is straightforward to impose limits on code size. Developing more sophisticated techniques that trade off code space for time is an open problem.

References

[1] O. Baby and J. Cohen. DNA parsing: a multi-pass constraint-based approach. *JFPLC*, 1996. Available at `http://www.cs.brandeis.edu/~obaby`.

[2] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In *International Logic Programming Symposium (ILPS)*, pages 457–471, 1994.

[3] M. Carlsson. On the efficiency of optimizing shallow backtracking in compiled Prolog. In *International Conference on Logic Programming (ICLP)*, 1989.

[4] S. Dawson. *Theory and practice of deterministic evaluation of logic programs*. PhD thesis, State University of New York at Stony Brook, December 1995.

[5] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting determinacy in logic programs. *In ICLP*, pages 424–438, 1993.

[6] S. Debray and N. Lin. Cost analysis of logic programs. *ACM TOPLAS*, 15(5):826–875, November 1993.

[7] S. Debray, P. Lopez-Garcia, and M. Hermenegildo. Non-failure analysis of logic programs. In *ICLP*, pages 48–62, 1997.

[8] S. Debray and D.S. Warren. Functional computations in logic programs. *ACM TOPLAS*, 11(3):451–481, July 1989.

[9] R. Giacobazzi and L. Ricci. Detecting determinate computations by bottom-up abstract interpretation. In *European Symposium on Programming*, pages 167–181, 1992.

[10] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the Mercury compiler. In *Nineteenth Australasian Computer Science Conference*, pages 337–346, 1996.

[11] T. Hickey and S. Mudambi. Global compilation of Prolog. *J. Logic Prog.*, 7:193–230, 1989.

[12] D.B. Kemp and P.J. Stuckey. Optimizing bottom-up evaluation for constraint queries. *J. Logic Prog.*, 26:1–30, January 1996.

[13] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction : Towards a maximal integration. In *Joint International Conference and Symposium on Logic Programming*, pages 319–332, 1996.

[14] J.W. Lloyd. *Foundations of Logic Programming, Second, Extended Edition*. Springer-Verlag, 1993.

[15] K. Mariott and P.J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. *In POPL*, pages 334–344, 1993.

[16] C.S. Mellish. Some global optimizations for a Prolog compiler. *J. Logic Prog.*, 2:43–66, 1985.

[17] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *POPL*, pages 414–427, 1997.

[18] K. Post. Mutually exclusive rules in logic programming. In *ILPS*, pages 472–486, 1994.

[19] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. *In TAPSOFT'87*, pages 111–125, March 1987.

[20] A. Roychoudhury, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Making success out of early failures. Technical report, State University of New York at Stony Brook, 1997.

[21] D. Sahlin. The mixtus approach to automatic partial evaluation of full Prolog. In *North American Conference on Logic Programming*, 1990.

[22] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog*. PhD thesis, Swedish Institute of Computer Science, March 1991.

[23] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[24] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. *In ICLP*, pages 200–207, 1985.

[25] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. *J. Logic Prog.*, 16:361–414, 1993.

[26] XSB. The XSB logic programming system v1.7, 1997. Available from `http://www.cs.sunysb.edu/~sbprolog`.

[27] N. Zhou, T. Takagi, and K. Ushijima. A matching tree oriented abstract machine for Prolog. *In ICLP*, pages 159–173, 1990.